CS2315
Algorithm Fundamentals

Searching Algorithms

Lectures prepared by: Dr. Manal Alharbi, and Dr. Areej Alsini

Review: Searching Problem

- Assume A is an array with n elements A[1], A[2], … A[n]. For a given element x, we must determine whether there is an index *j;* $1 \leq j \leq n$, such that $x = A[j]$

- Two algorithms, among others, address this problem
  - Linear Search
  - Binary tree: Binary Search

Linear Search Algorithm

Algorithm: LINEARSEARCH

Input: array $A[1..n]$ of n elements and an element x.

Output: j if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

1. $j = 1$
2. **while** $(j \leq n)$ ***and*** $(x \neq A[j])$ **do**
3. $j = j + 1$
4. **end while**
5. **if** x=A[j] **then return** j **else return** 0

Analyzing Linear Search

- One way to measure efficiency is to count how many statements get executed before the algorithm terminates
- One should keep an eye, though, on statements that are executed “repeatedly”.
- What will be the number of “element” comparisons if x
  - First appears in the first element of A
  - First appears in the middle element of A
  - First appears in the last element of A
  - Doesn’t appear in A.

Analyzing Linear Search

- One way to measure efficiency is to count how many statements get executed before the algorithm terminates
- One should keep an eye, though, on statements that are executed “repeatedly”.
- What will be the number of “element” comparisons if x
  - First appears in the first element of A — 1 comparison
  - First appears in the middle element of A — $\frac{n}{2}$ comparisons
  - First appears in the last element of A — n comparisons
  - Doesn’t appear in A. — n comparisons

Analyzing Linear Search

We are interested here in **finding the number of element comparisons .**In the algorithm, element comparisons are done in the while statement (Step 2) of the algorithm, that is

2. while ($j \leq n$ ***and*** $x \neq A[j]$

The minimum number of comparisons occurs when **the element** we are looking for **is the first element in the array**. In this case, there is only one element comparison.

The maximum number of element comparisons occurs when **the element is not among the elements of the array or in the last index**. This results in n comparisons.

Thus, the number of element comparisons in the linear search algorithm is between 1 and n inclusive.

Linear Search Algorithm

Algorithm: LINEARSEARCH

Input: array $A[1..n]$ of n elements and an element x.

Output: j if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

Algorithm	Cost	Time
1. $j = 1$	c_1	1
2. **while** ($j \leq n$) ***and*** ($x \neq A[j]$) **do**	c_2	K
4. $j = j + 1$	c_3	$K - 1$
5. **end while**		
6. **if** x=A[j] **then**	c_6	1
7. **return** j	c_7	1
8. **else return** 0	c_8	1

- K ranges between 1 and n.
- Worst-case time = $n + n - 1 = 2n - 1$
- Total time complexity = $\boldsymbol{O(2n - 1) = O(n)}$

Can we do better? Review: binary tree- concept

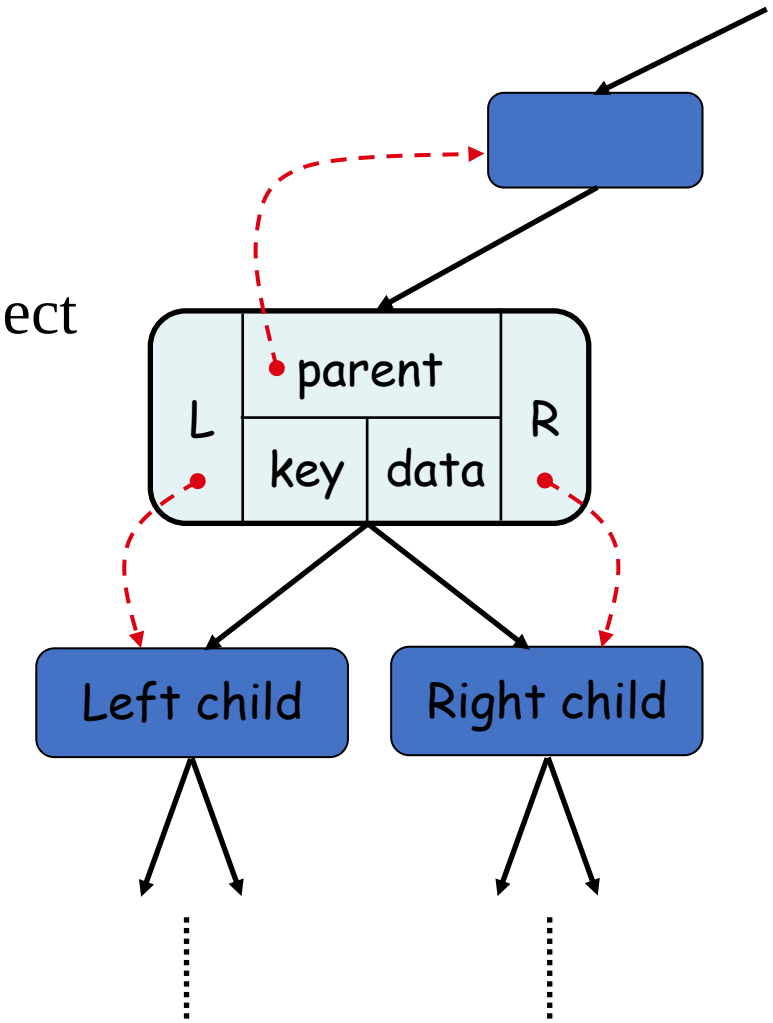
- Yes, using **binary search**.

- **Algorithms Design: Divide-and-conquer**
  - Let π be any problem with $|\pi| = n$. To solve using divide-and-conquer the following steps are involved:
    - Generate k subproblems form π (for some $k \geq 1$).
      - Let these subproblems be $\pi_1, \pi_2, \ldots\ldots, \pi_k$;
    - for $i = 1$ to k do
      - Recursively (or otherwise) solve π_i;
    - Combine the solutions obtained in step 2 to create a solution for π.

  - A classical example of divide-and-conquer is **binary search**. For this problem, the input are a sorted sequence $X = k_1, k_2, \ldots\ldots, k_n$ and another element x. The problem is to check if $x \in X$.

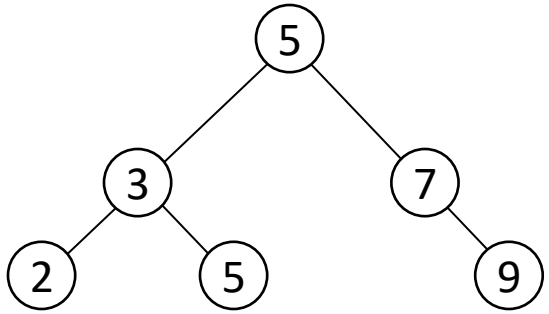
Binary Search Trees- Review

- Tree representation:
  - A linked data structure in which each node is an object
- Node representation:
  - `Key` field
  - `Left`: pointer to left child
  - `Right`: pointer to right child
  - `p`: pointer to parent (`p [root [T]] = NIL`)
- Satisfies the binary-search-tree property !!



Binary Search Tree Property

- Binary search tree property:
  - If y is in left subtree of x,
    then key [y] ≤ key [x]
  - If y is in right subtree of x,
    then key [y] ≥ key [x]

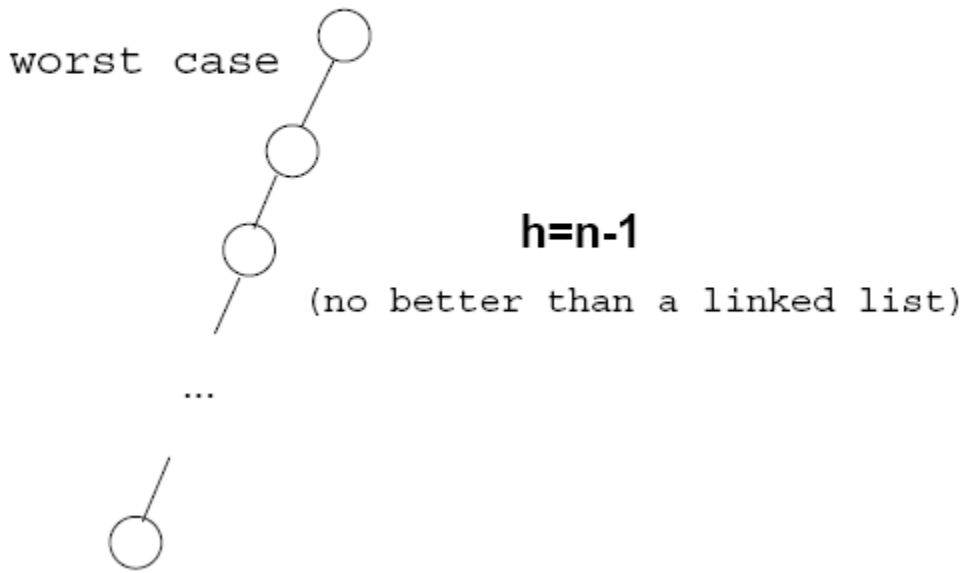


Binary Search Trees

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: $\Theta(\log n)$
    - The expected height of the tree is $\log n$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes

Worst Case

- If the tree is very **unbalanced**, then running time will be $O(n)$.



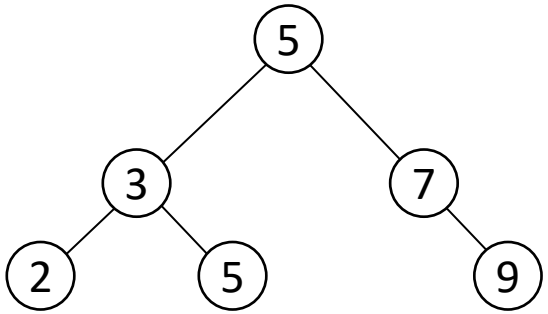
Traversing a Binary Search Tree

- **Inorder** tree walk:
  - Root is printed between the values of its left and right subtrees left, root, right
  - Keys are printed in sorted order

- **Preorder** tree walk:
  - root printed first: root, left, right

- **Postorder** tree walk:
  - root printed last: left, right, root

Example:



Inorder: 2 3 5 5 7 9

Preorder: 5 3 2 5 7 9

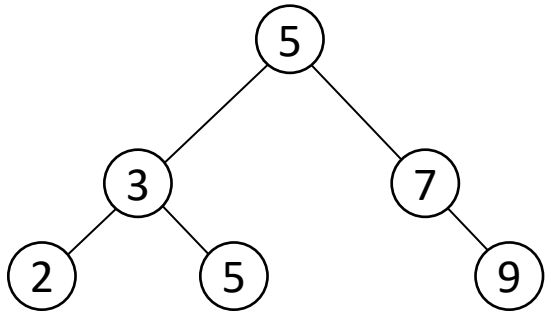
Postorder: 2 5 3 9 7 5

Traversing a Binary Search Tree

Alg: INORDER-TREE-WALK(x)

1. **if** x ≠ NIL
2. **then** INORDER-TREE-WALK (left [x])
3. print key [x]
4. INORDER-TREE-WALK (right [x])

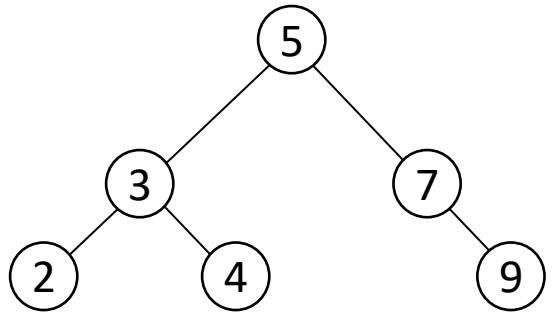
- *E.g.:*



Output: 2 3 5 5 7 9

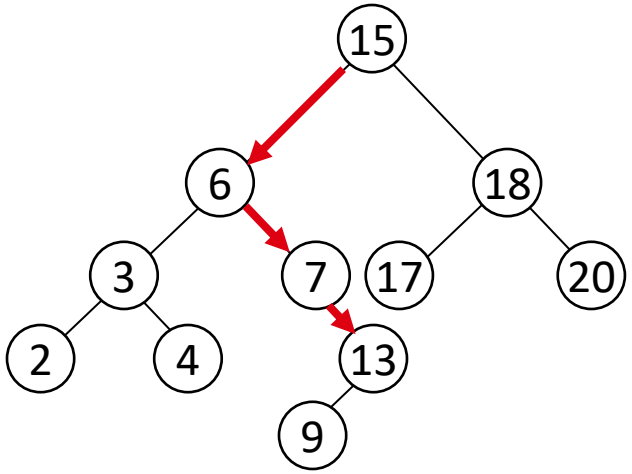
- Running time:
  - Θ(n), where n is the size of the tree rooted at x

Searching for a Key



- Given a pointer to the root of a tree and a key `k`:
  - Return a pointer to a node with key `k`
    if one exists
  - Otherwise return NIL
- Idea
  - Starting at the root: trace down a path by comparing `k` with the key of the current node:
    - If the keys are equal: we have found the key
    - If `k < key[x]` search in the left subtree of `x`
    - If `k > key[x]` search in the right subtree of `x`

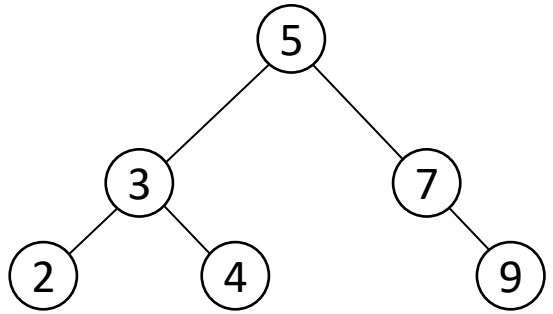
Example: TREE-SEARCH



- Search for key 13:
  - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a Key

Alg: TREE-SEARCH(x, k)



1. **if** x = NIL or k = key [x]
2. **then return** x
3. **if** k < key [x]
4. **then return** TREE-SEARCH(left [x], k)
5. **else return** TREE-SEARCH(right [x], k)

Running Time: *O* (h),
h – the height of the tree

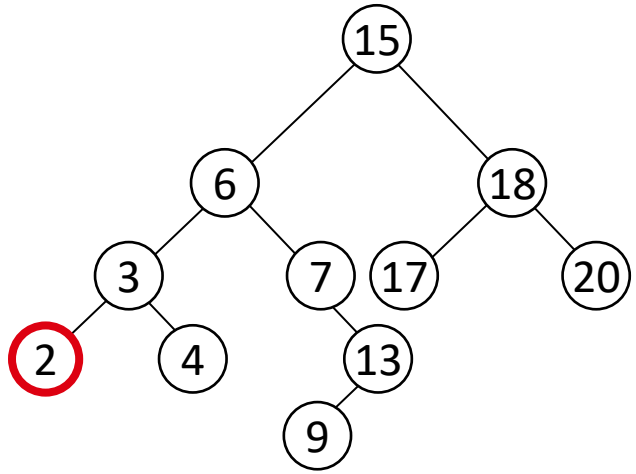
Finding the Minimum in a Binary Search Tree

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

Alg: TREE-MINIMUM(x)

1. **while** left [x] ≠ NIL
2. **do** x ← left [x]
3. **return** x

Running time: O(h), h – height of tree



Minimum = 2

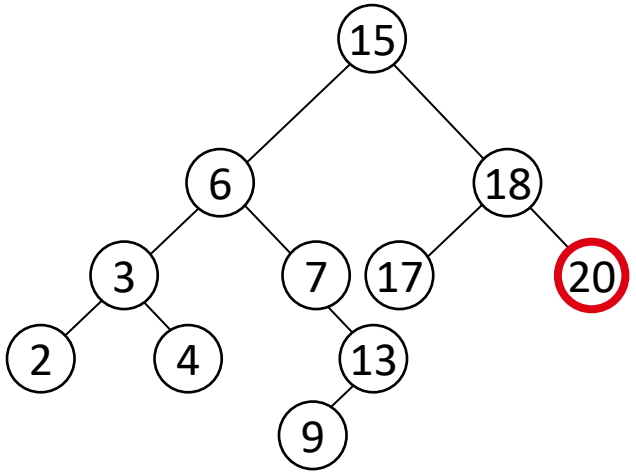
Finding the Maximum in a Binary Search Tree

- Goal: find the maximum value in a BST
  - Following right child pointers from the root, until a NIL is encountered

Alg: TREE-MAXIMUM*(x)*

1. **while** right [x] ≠ NIL
2. **do** x ← right [x]
3. **return** x

- Running time: O(h), h – height of tree

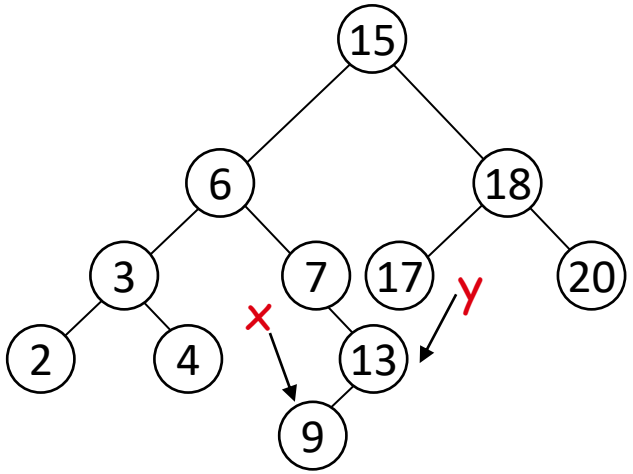


Maximum = 20

Successor

Def: *successor* (x) = y, such that key [y] is the smallest key > key [x]

- *E.g.:* *successor (15)* = *17*
  successor (13) = *15*
  successor (9) = *13*

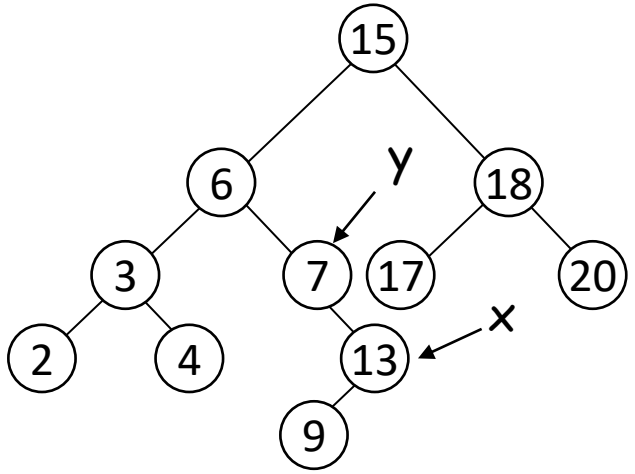


- Case 1: right (x) is nonempty
  - *successor* (x) = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: *successor* (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

Finding the Successor

Alg: TREE-SUCCESSOR*(x)*

```
1.  if right [x] ≠ NIL
2.     then return TREE-MINIMUM(right [x])
3.  y ← p[x]
4.  while y ≠ NIL and x = right [y]
5.     do x ← y
6.        y ← p[y]
7.  return y
```

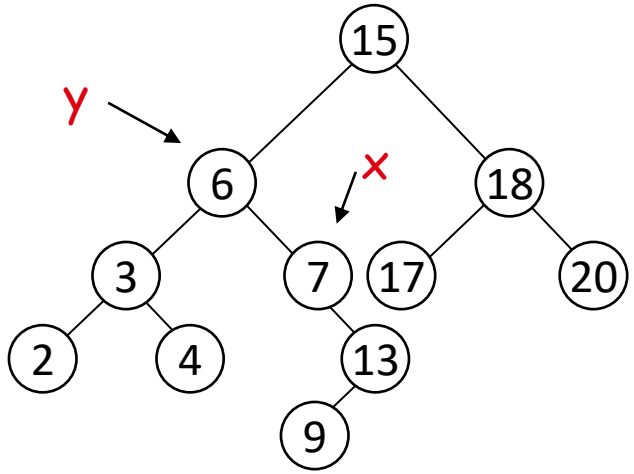


Running time: $O(h)$, h – height of the tree

Predecessor

Def: *predecessor* (x) = y, such that `key [y]` is the biggest `key < key [x]`

- *E.g.:* *predecessor (15)* = *13*
  predecessor (9) = *7*
  predecessor (7) = *6*



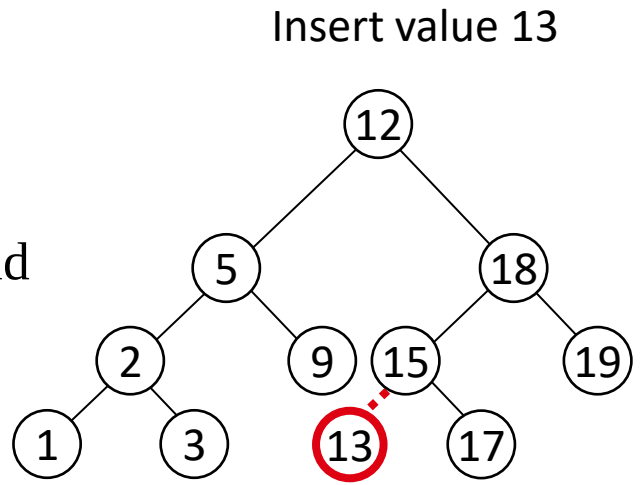
- Case 1: `left (x)` is nonempty
  - *predecessor* (x) = the maximum in `left (x)`
- Case 2: `left (x)` is empty
  - go up the tree until the current node is a right child:
    predecessor (x) is the parent of the current node
  - if you cannot go further (and you reached the root):
    `x` is the smallest element

Insertion

- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If key [x] < v move to the right child of x,

    else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < key [y] insert the new node as y's left child

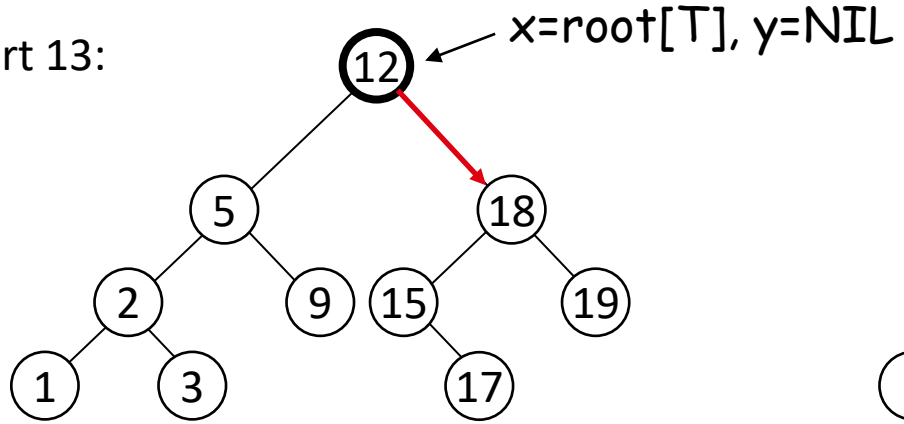
    else insert it as y's right child

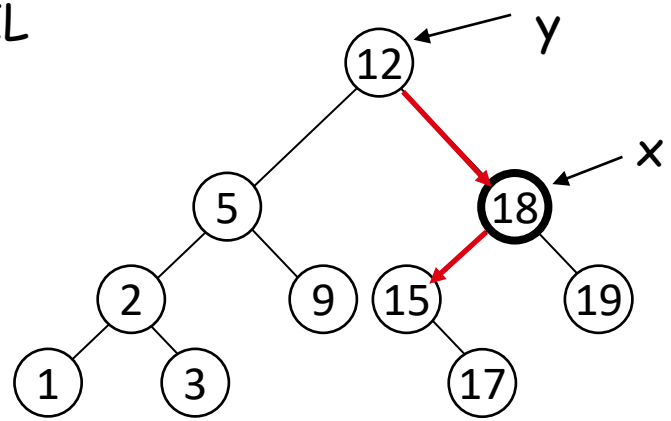


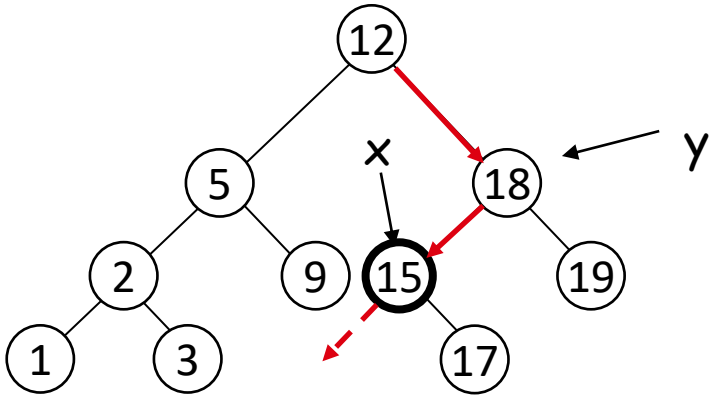
  - Begining at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x (“trailing pointer”)

Example: TREE-INSERT

Insert 13:

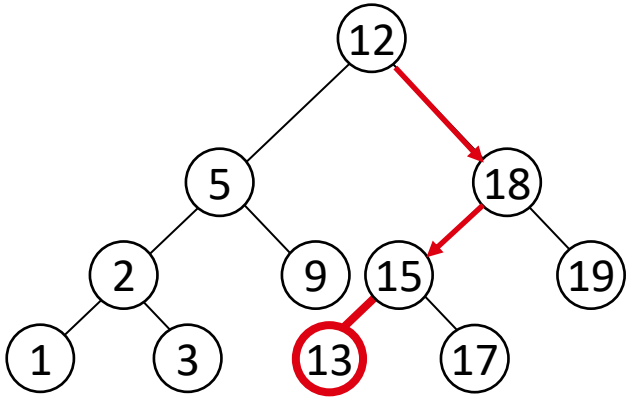






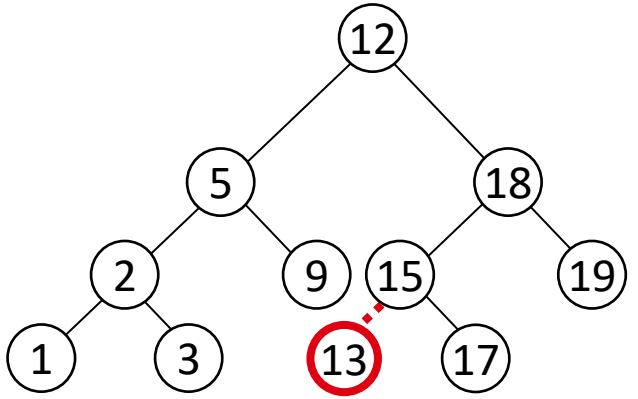
x = NIL

y = 15



Alg: TREE-INSERT(T, z)

```
1.  y ← NIL
2.  x ← root [T]
3.  while x ≠ NIL
4.      do y ← x
5.          if key [z] < key [x]
6.              then x ← left [x]
7.              else x ← right [x]
8.  p[z] ← y
9.  if y = NIL
10.     then root [T] ← z
11.     else if key [z] < key [y]
12.             then left [y] ← z
13.             else right [y] ← z
```

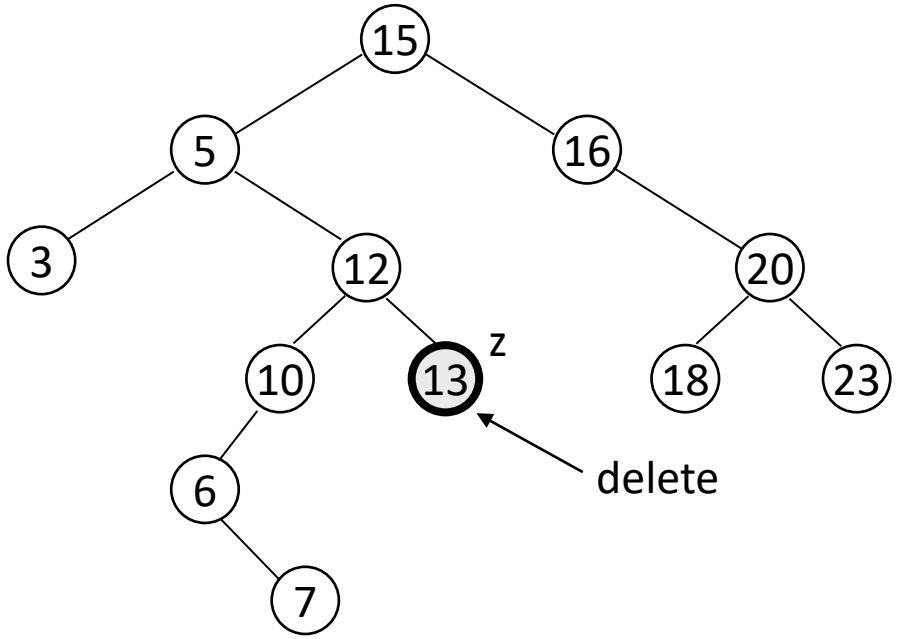


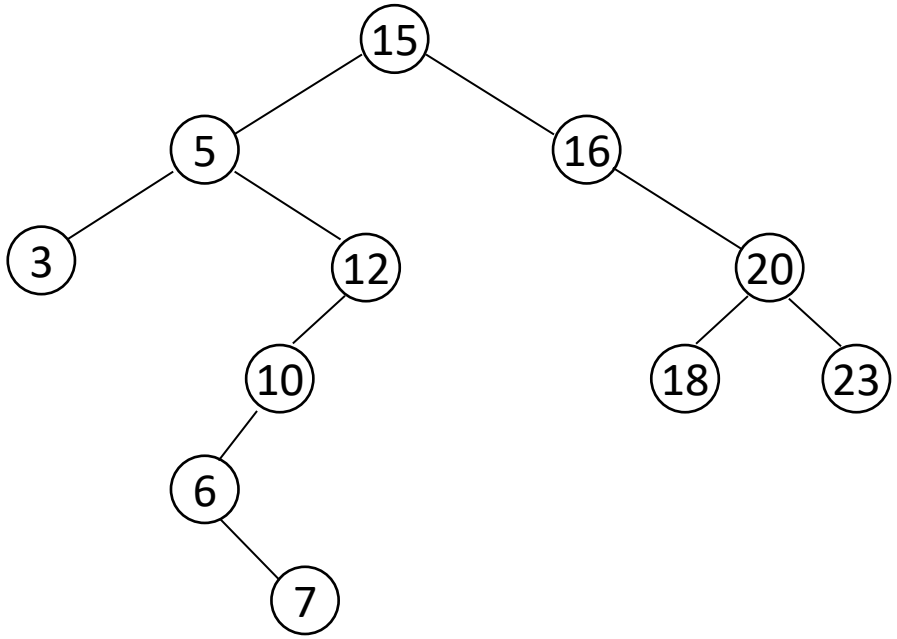
Tree T was empty

Running time: $O(h)$

Deletion

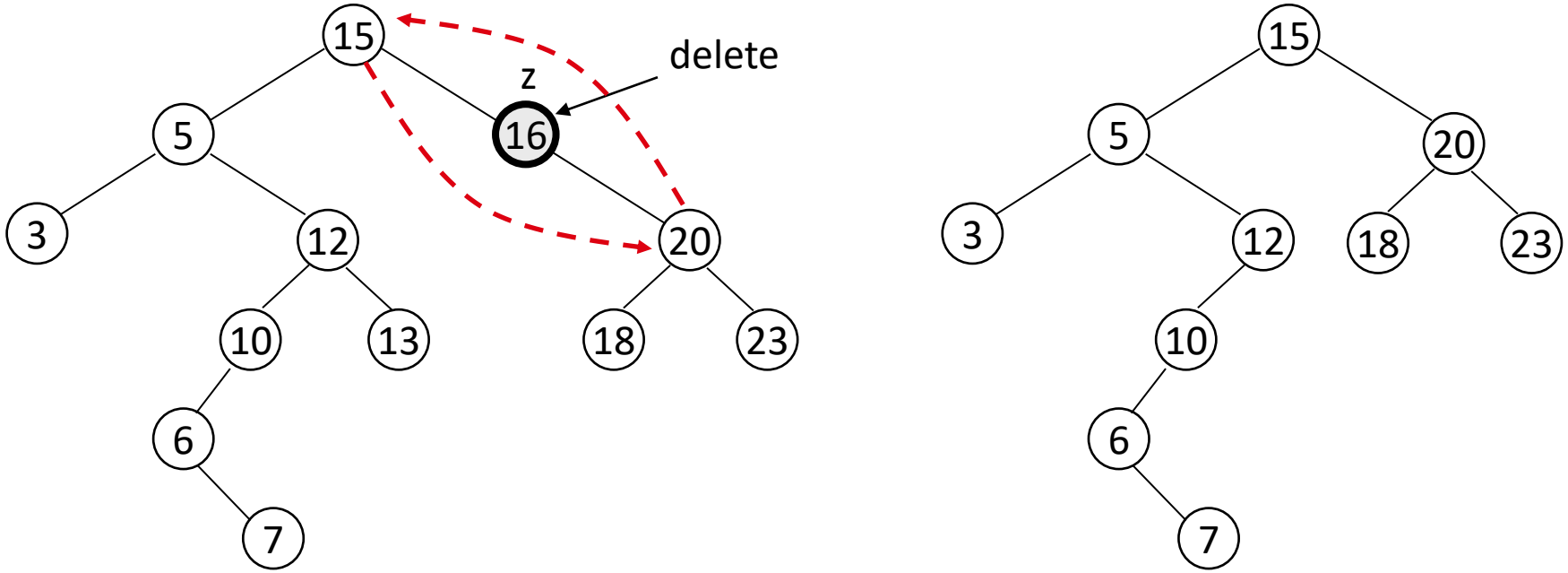
- Goal:
  - Delete a given node **z** from a binary search tree
- Idea:
  - **Case 1: z** has no children
    - Delete **z** by making the parent of **z** point to NIL





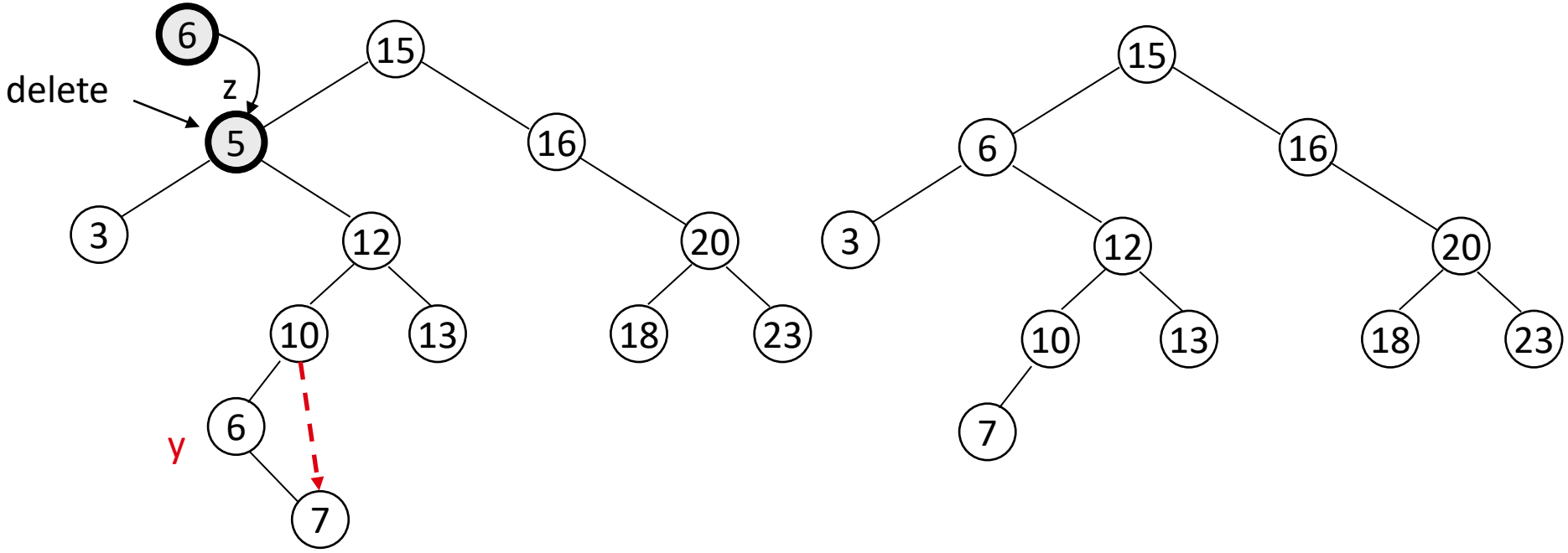
Deletion

- **Case 2: `z` has one child**
  - Delete `z` by making the parent of `z` point to `z`'s child, instead of to `z`



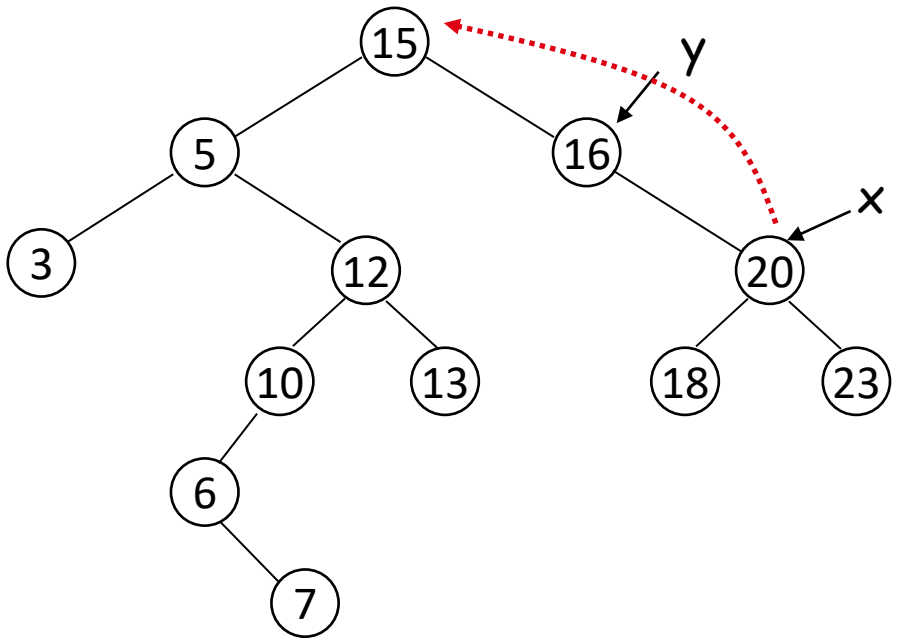
Deletion

- **Case 3: z** has two children
  - **z**'s successor (y) is the minimum node in **z**'s right subtree
  - **y** has either no children or one right child (but no left child)
  - Delete **y** from the tree (via Case 1 or 2)
  - Replace z's key and satellite data with y's.



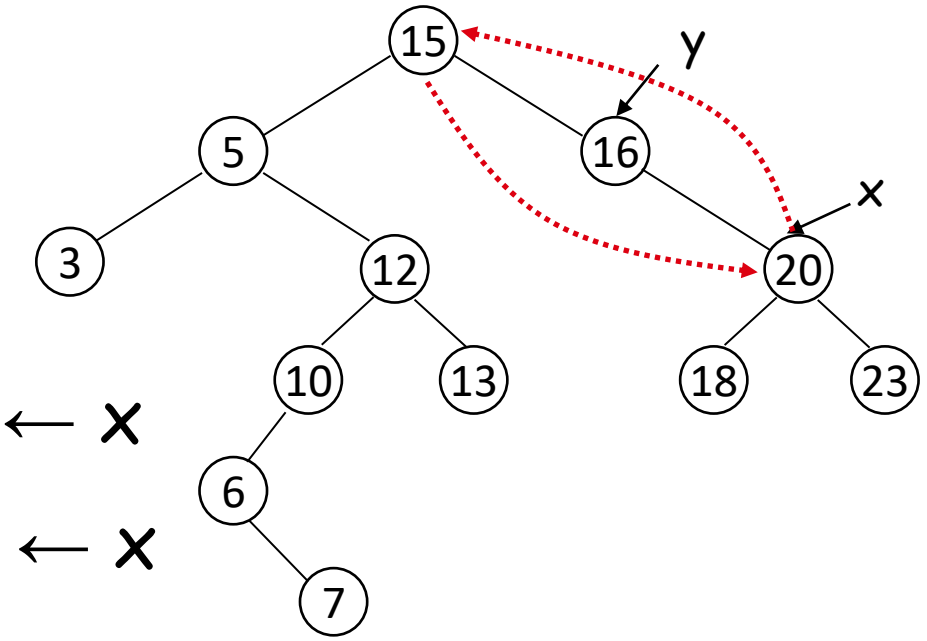
TREE-DELETE(T, z)

1. **if** left[z] = NIL or right[z] = NIL
2. **then** y ← z — z has one child
3. **else** y ← TREE-SUCCESSOR(z) — z has 2 children
4. **if** left[y] ≠ NIL
5. **then** x ← left[y]
6. **else** x ← right[y]
7. **if** x ≠ NIL
8. **then** p[x] ← p[y]



TREE-DELETE(T, z) – cont.

```
9.   if p[y] = NIL
10.     then root[T] ← x
11.     else if y = left[p[y]]
12.             then left[p[y]] ← x
13.             else right[p[y]] ← x
14.  if y ≠ z
15.     then key[z] ← key[y]
16.          copy y's satellite data into z
17.  return y
```

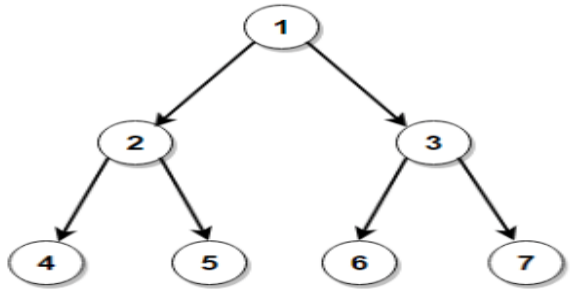


Running time: $O(h)$

Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$
- These operations are fast if the height of the tree is small – otherwise their performance is similar to that of a linked list

Theorem



- **The number of comparisons performed by Algorithm BINARY SEARCH on a sorted array of size n is at most $\lfloor \log n \rfloor + 1$**

- **Proof:**
  - **Fact:** If T is a binary tree with n nodes whose height is h, then,

$$\underset{\text{max}}{n} \geq h \geq \underset{\text{min}}{\log(n+1)}.$$

  - In a binary tree the **maximum number of nodes** we can have **in level** i is 2^{i-1}.
  - This implies that the maximum number of nodes we can have in a binary tree of height h is $1 + 2 + 4 + \cdots + 2^h = 2^h - 1$.
  - In other words, $n \leq 2^h - 1$ ➔ $\log_2 n \leq h\text{-}1$ ➔ $h \geq \log_2 n + 1$
  - The fact that $h \leq n$ is easy to see. This happens when we have a skewed tree where each node (other than the leaf) has a single child.

- A binary tree is said to be *full* if every non-leaf has exactly two children and all the leaves are at the same level. A binary tree is defined to be *complete* if it is full except that there could be some nodes missing in the last level and the missing nodes in the last level (if any) are right justified. The height of a complete binary tree with n nodes is $\Theta(\log n)$.

Binary Search

- **We can do "better" than linear search if we knew that the elements of A are sorted, say in non-decreasing order (increasing order).**
- As stated above, we are interested in determining whether a given element is among the list of elements stored in the array of size n.
- When the elements are sorted, we use a more efficient algorithm such as the binary search algorithm as follows:
  - Let *A[low.. high]* be a non-empty array of elements sorted in nondecreasing order and let *A[mid]* be the middle element.
  - The idea is that you can compare x to the middle element of A, say A[middle].
    - If *x> A[mid]:*
      - We observe that If *x* is in *A*, then it must be one of the elements *A[mid+1], A[mid+2],…., A[mid+ high]*
      - It follows that we only need to search for in *A[mid+1… high].*
    - In other words, the entries in *A[low…mid]* are discarded.
    - Similarly, if x< *A[mid]*, then we only need to search for in *A[low…mid-1]*
- This results in an efficient strategy, which is referred to as binary search.

Example 1. In this example, we will search for the element x = 35 or any value > 35 in an array of 14 elements. The array is assumed to be sorted in nondecreasing order with the values as shown.

Index:	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A[1..14] =	1	4	5	7	8	9	10	12	15	22	23	27	32	35

The execution of the binary search algorithm on the above array proceeds as follows:

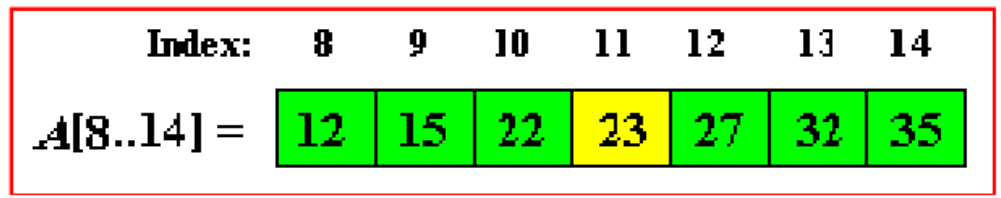
First, the index (referred to as mid in Algorithm BINARYSEARCH) of the array element to be compared with the value x is computed using the equation:

$$mid = \lfloor (low + high)/2 \rfloor = \lfloor (1 + 14)/2 \rfloor = 7$$

and the value x (=35) is compared with A[7] (= 10) and since 35 > A[7], the portion of the array A[1 .. 7] is discarded.

Example 1 (Cont.)

Thus, the remaining portion of the array to be searched next is now reduced to the following subarray:



Next, we repeat the process of computing the index of the array element (among the remaining elements of the subarray A[8..14]) to be compared with x. In this case, we use low = 8 and high =14, and get:

$$mid = \lfloor (low + high)/2 \rfloor = \lfloor (8+14)/2 \rfloor = 11$$

Then x is compared with A[11] (= 23) and since 35 > A[11], the portion of the array A[8..11] is discarded.

- **Example 1 (Cont.)**
  - Finally, x is compared with A[14] (=35) as

$$mid = \lfloor (14 + 14)/2 \rfloor = 14$$

and hence the search is successively completed.

Binary Search Algorithm

Algorithm: BINARY_SEARCH

Input: An array $A[1..n]$ of n elements sorted in non-decreasing order and an element x.

Output: j if $x = A[j]$, $1 \leq j \leq n$, and 0 otherwise.

```
1. low = 1; high = n; j = 0
2. while (low ≤ high) and (j = 0)
3.    mid = ⌊(low + high)/2⌋
4.    if x = A[mid] then j = mid
5.    else if x < A[mid] then high = mid – 1
6.    else low =  mid + 1
7. end while
8. return j
```

Can be considered as one comparison